Des Autom Embed Syst (2016) 20:311–339
DOI 10.1007/s10617-016-9179-z

A hierarchical run-time adaptive resource allocation framework for large-scale MPSoC systems

Wei Quan[1,2] · **Andy D. Pimentel**[1]

Received: 25 November 2015 / Accepted: 22 September 2016 / Published online: 1 October 2016

Abstract In the embedded computer system domain, MPSoC systems have become increasingly popular due to the ever-increasing performance demands of modern embedded applications. The number of processing elements in these MPSoCs also steadily increases. Whereas current MPSoCs still contain a limited number of processing elements, future MPSoCs will feature tens up to hundreds of (heterogeneous) processing elements that are all integrated on a single chip. On these future large-scale MPSoC systems, the mapping of applications onto the hardware resources plays an important role to fully explore the parallelism of applications. In this article, a hierarchical run-time adaptive resource allocation framework which uses an intelligent task remapping approach is proposed to improve the system performance for large-scale MPSoCs.

Keywords Embedded systems · KPN · MPSoC · Task mapping · Simulation

1 Introduction

The ever-increasing performance requirements of embedded applications stimulate the development of MPSoC systems in the embedded systems domain. These MPSoC systems, such as most of current smart-phones, digital televisions, set-tops, etc., are often heterogeneous systems containing programmable processor cores for flexible application support as well as dedicated processing elements for achieving power and performance goals. The number of processing elements in these MPSoCs also steadily increases because of the never-ending

✉ Wei Quan
quanwei02@gmail.com

Andy D. Pimentel
a.d.pimentel@uva.nl

1 University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands

2 Present Address: National University of Defense Technology, Yanwachi Main Street 47, Changsha 410073, Hunan, China

performance demands of embedded applications. Whereas current MPSoCs still contain a limited number of processing elements, future MPSoCs will feature tens up to hundreds of (heterogeneous) processing elements that are all integrated on a single chip to handle the next generation of embedded applications like real-time physics, artificial intelligence, 3D rendering effects and so on [14].

Besides the performance requirements, the dynamism and complexity of application workloads executed on embedded systems are also rapidly increasing. Today's MPSoC systems often require supporting a growing number of applications and standards, where multiple applications can run simultaneously. For each single application, there may also be different execution modes (or program phases) with different computational and communication requirements. For example, in Software Defined Radio appliances a radio may change its behaviour according to resource availability, such as the Long Term Evolution (LTE) standard which uses adaptive modulation and coding to dynamically adjust modulation schemes and transport block sizes based on channel conditions. As a consequence, the behaviour of application workloads executing on future embedded systems can change dramatically over time. To improve the Quality of Service (QoS like performance, precision, energy consumption, etc.) for future embedded systems under such dynamic and complex application workloads, *system adaptivity* is very important. Future embedded systems will need to continuously customize their underlying system at run time according to the application workload at hand and the state of the system itself.

The combination of the above trends in both application and architecture of embedded systems lead to the research of this work on *adaptive large-scale MPSoC-based embedded systems*. There are a number of technology enablers toward adaptive systems. For example, there has been significant research attention on techniques for dynamically re-mapping application tasks to different processing resources [2,3,16]. But they are rarely considered for large-scale heterogeneous MPSoCs. Other examples are techniques for dynamically changing system parameters such as done in dynamic frequency and voltage scaling, and techniques for dynamically reconfiguring processing components for accelerating application tasks (e.g., [5,43]) or network components to customize the network to a specific application workload (e.g. [42]). However, these techniques need additional hardware supports on the system.

The design of future large-scale MPSoC systems is still an open research question. A very popular prototype is the tile-based scalable system [8,19,20,31,41,44]. In this work, we consider the tile-based MPSoC system as our target system. To increase the adaptivity of future tile-based large-scale heterogeneous MPSoC systems, we propose a Scenario-based Hierarchical run-time Adaptive Resource Allocation (SHARA) framework. This framework takes advantages of the state-of-the-art solutions for modern adaptive heterogeneous MPSoC systems where the system adaptivity is achieved by adaptively adjusting the mapping of applications to the underlying hardware resources which is optimised at design time. More specifically, we reconsider the method in the state-of-the-art solutions that allows for dynamically reconfiguring the system at run time based on pre-optimized system configurations, such as task mappings derived at design time [17,25,27,29,31,37,45], and extend it by solving the issues of *scalability* as introduced in [25] and *blind adaptivity* (a system reconfiguration that should not have happened because of its large overhead actually happened) that are usually existed in these solutions for our target large-scale heterogeneous MPSoC system.

The contributions of this work can be summarized as follows:

(1) Traditionally, run-time managers are either centralized or distributed. However, as a centralized approach comes with a performance bottleneck and a distributed approach leads to a high complexity, both approaches do not fulfill the requirements of large-scale embedded systems [31]. To overcome this problem, a hierarchical resource management mechanism is

implemented in our framework where a global resource manager takes charge of the workload distribution among tiles and the local resource manager in each tile optimises the resource allocation for the assigned applications.

(2) To handle the complex and dynamic application workloads for the target MPSoC system, we propose a hybrid approach for mapping applications to the underlying resources. Similar to the above-mentioned state-of-the-art solutions for modern adaptive MPSoC systems, there are two stages in our approach as well. The first stage is the design-time preparation stage where application mappings are optimised by our Design Space Exploration (DSE) approach. The second stage is the run-time mapping re-optimisation stage. Our proposed hybrid approach is able to solve the scalability problem of most existing solutions.

(3) For the purpose of avoiding a *blind* system adaptivity as mentioned above, a self-adaptive scheduler is presented for adaptivity throttling. The scheduler tries to predict whether or not reconfiguration of the system actually is beneficial when system execution environment changed. According to the prediction, the system will either be reconfigured or not. It is able to improve the system's efficiency as compared to MPSoCs that do not provide such intelligent reconfiguration control.

The remainder of this article is organized as follows. Section 2 gives some prerequisites for this article. Section 3 describes the hierarchical resource management mechanism on the target MPSoC system and Sect. 4 provides a detailed description of our SHARA framework. Section 5 introduces the experimental environment and presents the results of our experiments. Section 6 discusses related work, after which Sect. 7 concludes the article.

2 Prerequisites

2.1 Application and architecture model

In this work, we target the multimedia application domain. For this reason, we use the Kahn Process Network (KPN) model of computation [12] to specify application behaviour since this model of computation fits well to the streaming behaviour of multimedia applications. In a KPN, an application is described as a network of concurrent processes that are interconnected via FIFO channels. This means that an application can be represented as a directed graph $KPN = (P, F)$ where P is set of processes (tasks)[1] p_i in the application and $f_{ij} \in F$ represents the FIFO channel between two processes p_i and p_j.

As introduced in the previous section, the workload of MPSoC systems are increasingly dynamic. Here, we use the concept of scenario [6,22] to capture the application dynamism on our target MPSoC system. There are two different kinds of scenarios: inter-application scenarios describe the simultaneously running applications in the system, while intra-application scenarios define the different execution modes for each application. The combination of these inter- and intra-application scenarios are called *workload scenarios* [27], and specify the application workload in terms of the different applications that are concurrently executing and the mode of each application as shown in Fig. 1. We denote S as the set of all possible workload scenarios for the target applications. For a number of n target applications where each application has m execution modes, the total number of possible workload scenarios in S is $(m+1)^n - 1$. Each workload scenario $s_i \in S$ is described as a set of KPN graphs, $s_i = (..., KPN_j^k, ...)$ where KPN_j^k is the graph of app_j^k (application j, mode k) that is active in scenario s_i. Combining the KPN graphs in a workload scenario, the graph of a whole

[1] We use the terms process and task interchangeably in this article.

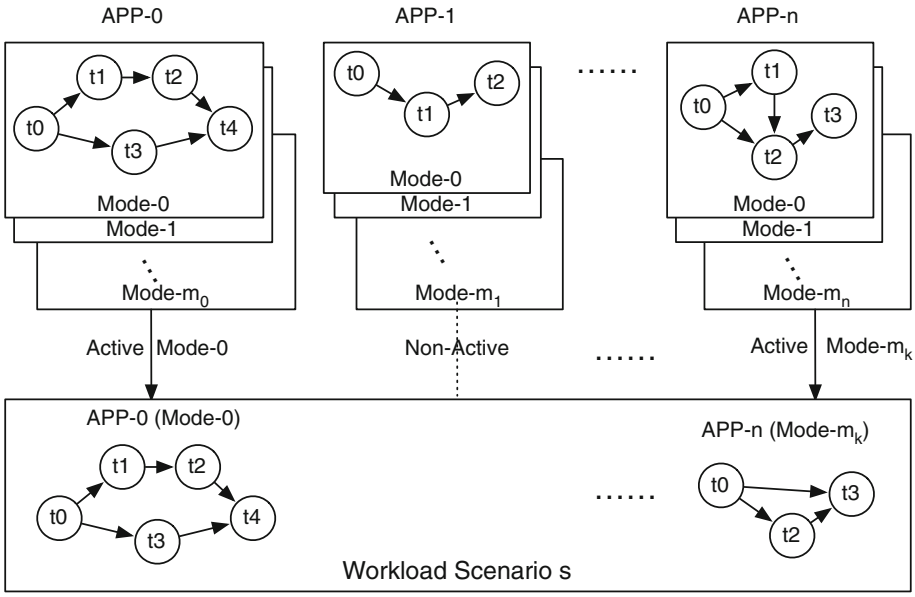

Fig. 1 Definition of a workload scenario

workload scenario can be expressed as $s_i = (T_i, C_i)$ where T_i is set of tasks in the scenario s_i and C_i represents the set of communication channel between two communicating tasks. Each element in T_i and C_i, noted as t_i^{km} and c_i^{kn} respectively, represents the $m-th$ task and the $n-th$ communication channel in application app_k which is active in workload scenario s_i.

Our target MPSoC architecture is illustrated in Fig. 2. This MPSoC is composed of four identical tiles. In each tile, there are four heterogeneous processing elements connected to a shared memory by bus. Note that the communication in our target system has multiple levels like the intra-processor communication by buffers, intra-tile communication by a shared bus and inter-tile communication by a simple mesh Network-on-Chip (NoC) similar to [31]. The reason for considering this type of MPSoC architecture is that the architecture of a tile in our target system can be designed by current state-of-the-art MPSoC design approaches like the work from [23]. Also this kind of layered architecture could reduce the communication congestion that might happen in a large-scale NoC. The target architecture of our large-scale MPSoC system can be modelled as a graph $MPSoC = (PE, M)$, where PE is the set of processing elements used in the architecture and M is a multiset of pairs $m_{ij} = (pe_i, pe_j) \in PE \times PE$ representing a buffered communication medium (like a Bus, NoC, etc.) between processors pe_i and pe_j. However, our proposed approach is not limited to the architecture we assumed here. It can be applied to other architectures as long as the system can be (virtually) divided into identical tiles.

2.2 Task mapping

The task mapping defines the binding of the components in a workload scenario (including the tasks and the communication channels) to the underlying architecture resources. Given a workload scenario and a target MPSoC, a correct mapping is a pair of unique assignments

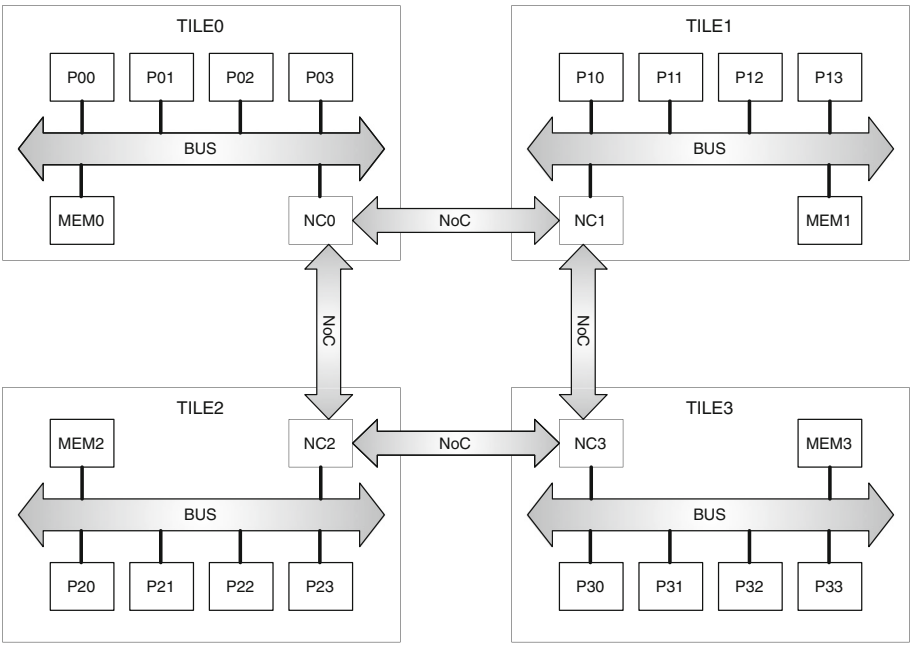

Fig. 2 The architecture of the target MPSoC system

$(\mu : T \rightarrow PE, \eta : C \rightarrow M)$ such that it satisfies $\forall c \in C, src(\eta(c)) = \mu(src(c)) \wedge dst(\eta(c)) = \mu(dst(c))$. For each workload scenario $s_i \in S$, the possible task mappings are denoted as TM_i with each single mapping $tm_i^j \in TM_i$ complying with the mapping constraint. In this work, we assume that the task mapping of applications on the target system only can be changed by task migration during system reconfiguration. Under these definitions, the computation cost of task $t_i^{km} \in T_i$ and the communication cost of $c_i^{kn} \in C_i$ in workload scenario s_i under the task mapping of tm_i^j is represented as et_{ij}^{km} and ec_{ij}^{kn} respectively.

Our objective in this work is to improve the system performance by adaptively reconfiguring the target system based on dynamically derived mappings for each detected workload scenario. It includes: firstly deriving a spatial and temporal optimised task mapping for each newly detected workload scenario on the target MPSoC system and secondly reconfiguring the system according to the newly derived mapping when the reconfiguration is predicted to be beneficial. Assuming a sequence of application scenarios need to be execution on the target system, our objective is to minimize the total execution time of the system, in other words, to maximize the throughput for target application scenarios.

3 Hierarchical resource management on the target MPSoC system

From the perspective of the control mechanism for system resource management, it can be divided into three categories [31,38]: (1) centralised resource management, (2) distributed resource management and (3) the combination of two previous methods. On modern MPSoC systems where a limited number of processing elements are present, centralised approaches

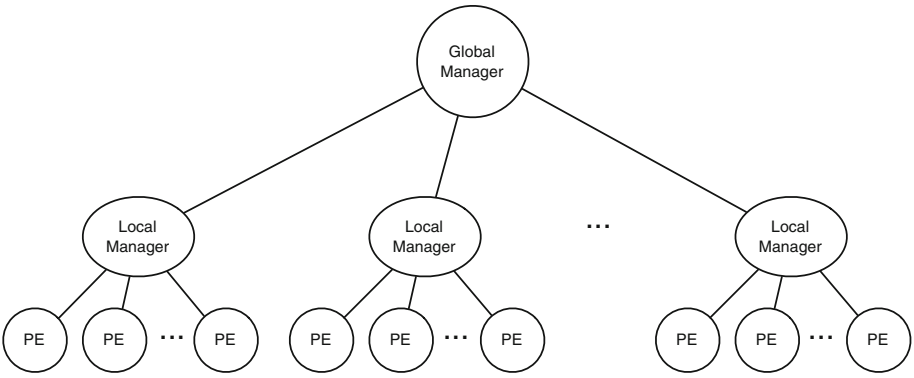

Fig. 3 The hierarchical resource management on the target MPSoC system

are usually considered because of their effectiveness and simplicity [18]. However, when the system scales, a centralised approach often suffers from its performance bottleneck as heavy communication might happen during resource reallocation when the number of processing elements is very large. To avoid this problem, distributed resource management approaches have been proposed [1,13,33]. However, these distributed approaches are usually complex and not easy to implement. Most importantly, this kind of approaches can only find a local optimal resource allocation solution. Consequently, a trade-off solution that combines the centralised and distributed resource management is commonly considered in multi-/many core systems [31]. For our target large-scale MPSoC system, we also adopt this hybrid approach where the control structure is hierarchically organised as in Fig. 3. The Global Manager (GM) takes charge of workload partition among tiles and each Local Manager (LM) optimises the resource allocation inside a tile for the assigned applications. This control mechanism can be implemented on the target system by either dedicated hardware (controller) or software. Currently, in the experiments of this work, we simulate our managers as hardware controllers with dedicated control channels. It means that the inter-controller communications will be done in dedicated communication channels with low overhead. Consequently, the system can response quickly by either reconfiguring the system or keeping unchanged when the workload scenario active on the target system changes. However, if these managers are implemented in software and directly executed on the processors in the target architecture by dividing tiles into a master tile and multiple slave tiles between tiles and a master processor and slave processors inside a tile. In this case, the communication between managers is done by the data channels between processors. Even though, this software implementation can save hardware resources comparing to the dedicated hardware managers, it will degrade the system performance because a larger overhead for computing new task mapping and a lower effective adaptivity throttling.

According to the above-mentioned control mechanism, a scenario-based hierarchical run-time adaptive resource allocation framework is proposed to adaptively reallocate the hardware resources on our target tile-based MPSoC systems. Note that this framework is not limited to the architecture we considered in this work, as tiles can be virtually divided on a target system. Taking a general NoC-based MPSoC system as illustrated in Fig. 4 as an example, the entire system can be firstly divided into identical tiles and then controlled by our approach. The details of our proposed SHARA framework will be explained in the following section.

To decide which implementation (hardware or software) of the above mentioned controllers is better for the target system, a designer should consider several factors like the

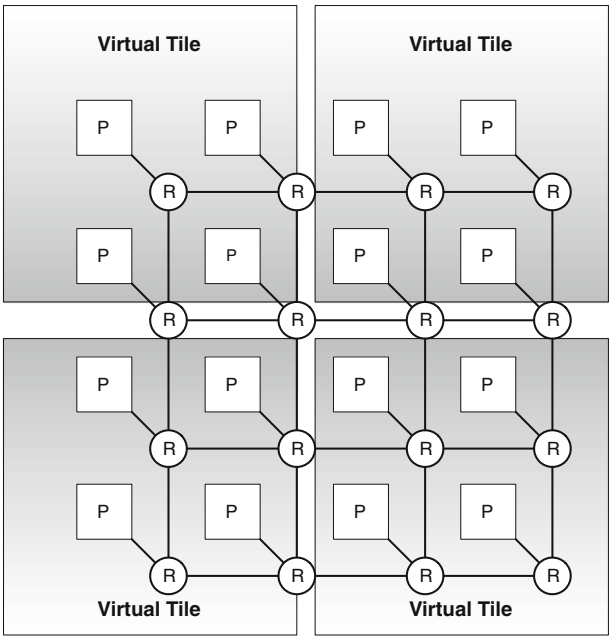

Fig. 4 An example of dividing a MPSoC into virtual tiles

costs, the performance requirements and the design complexity. For a system that needs to support hard real-time applications, hardware controllers with dedicated control channels are the best choice. As the unpredictable delay of software controllers (control channels and data channels are shared) will make the system unable to satisfy the hard real-time requirements. In our framework, the global manager takes charge of workload partition among tiles according to our tile-level load balance algorithm using the static mapping information. The control logic of this global manager is quite simple. The memory usage for storing the static mapping information explored at design time depends on the size of the target applications and architecture. Normally, hundreds of kilo bytes are big enough for storing the mapping information. The local manager of our framework uses the static mapping and the application/architecture related information (e.g. task execution time on a processors) to do mapping optimisation by our processor-level mapping algorithm. The complexity of this algorithm is much higher (2 orders of magnitude) than the algorithm adopted in the global manager. For the hardware implementation of a local manager, the resource usage will be higher compared to a global manager but far more less than a real processor. The memory consumption of a local manager is also larger than the global manager as more information needs to be stored, which means a few mega bytes maybe required. To keep up with the processor execution without introducing too much overhead, a local manager should be better to keep a similar frequency with the processors on the system and the global manager can have a lower operating frequency (1 order of magnitude). Even though the complexity of a local manager is far more less than a processor, keeping its frequency similar to the processors on the system can save a lot of time for the global manager. As the global manager needs local managers to calculate the mapping performance during its workload partition process.

4 Scenario-based hierarchical run-time adaptive resource allocation framework

When considering dynamic resource reallocation on a MPSoC system, three steps are needed. The first step is to decide when the resources reallocation should be triggered. For example, it could be a scenario change, a different QoS requirement, a system fault and so on. The second step is to derive a new resource scheme based on the detected trigger. After that, the third step is the actual system reconfiguration. The system reconfiguration in this work only changes the mappings of tasks (both computational and communicating tasks) to the hardware components in the target MPSoC. Figure 5 shows a high-level workflow of our SHARA framework. In this work, the trigger of the resource allocation events is the change of workload scenario on the target MPSoC system. At run time, the GM will continuously monitor the execution of workload scenarios on the target MPSoC system. When a new workload scenario is detected, the system will enter the resource reallocation stage. In this stage, the GM will try to redistribute the applications in the detected workload scenario according to the utilisation of each tile and the resource usage of each application in the system. Based on the new workload distribution and the potential reconfiguration benefit, the global scheduler in the GM will start a global workload scheduling. In each tile, if the workload (applications) allocated by the GM is different with its previous workload, the local scheduler of the LM will adaptively reconfigure the hardware resources based on the mapping optimised in the LM and the corresponding reconfiguration benefit.

In this article, we mainly focus on the last two steps of dynamic resource reallocation on our target MPSoC system as mentioned above. To derive a new mapping for a workload scenario on the target MPSoC system, we propose a scalable run-time task mapping approach which hierarchically maps the applications onto the tile-based MPSoC system. Normally, after deriving a new mapping for the new workload scenario, system reconfiguration should be done by the resource schedulers under the new generated mapping scheme. However, this is not always beneficial, like a very short execution duration for the new workload scenario, as its unignorable cost which in our case mainly comes from the task migration on the target system can't be ignored at run time. If the schedulers still reconfigure the target system when it is not beneficial, this will lead to the problem of *blind adaptivity* as introduced at the start of this article. To solve this problem, we propose an *adaptivity throttling* technique in the resource schedulers of the system which is able to smartly decide whether a system reconfiguration is suitable.

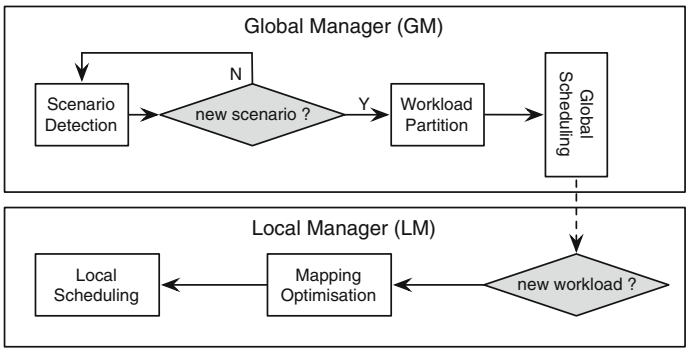

Fig. 5 A high-level workflow of SHARA framework

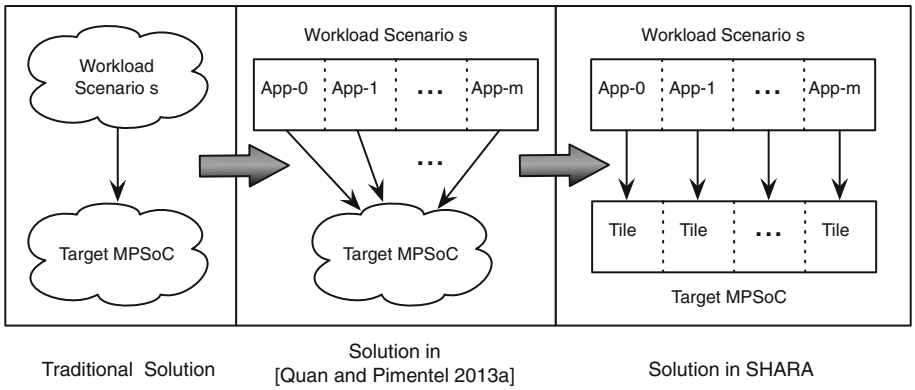

Fig. 6 Divide-and-conquer solution for complex task mapping problem

4.1 Scalable run-time task mapping in SHARA

To overcome the shortcomings of pure static [4,11,15] and dynamic [21,36] task mapping algorithms, hybrid (semi-static) approaches have become increasingly popular in recent years. Usually, in this kind of approaches, multiple mapping solutions are found at design time and applied at run time based on the current state of the system. However, these methods typically still suffer from scalability issues when the number of workload scenarios becomes very large as they need to find and store one or more optimal task mappings per scenario at design time (to be used at run time). One solution to address this problem is by reducing the number of workload scenarios by means of clustering [6,27]. However, these methods still suffer from an additional problem of searching for optimal mappings of (clustered) workload scenarios at design time: it should already been known at design time which applications can execute on the target platform. This implies that extending the system with a new application would require to redo the entire design-time mapping preparation for all (clustered) workload scenarios.

In our SHARA framework, we address the complex task mapping problem on our tile-based heterogeneous MPSoC system using the idea of a hybrid task mapping technique proposed in our previous work [25] which prepares *partial task mappings* for workload scenarios at design time and completes the mappings for the entire scenario at run time. This task mapping approach was proposed for a small-scale heterogeneous MPSoC system. It solves the scalability problem with regard to the number of tasks in the mapping problem. However the complexity of a task mapping problem not only depends on the number of application tasks but also on the number of target processing elements[2]. In this work, we solve this problem by taking advantage of both the MPSoC architecture and its hierarchical control mechanism as shown in Fig. 6. For our tile-based MPSoC system, we made a few assumptions for our task mapping approach. An entire application can only be mapped to a single tile to reduce the communication overhead between tasks inside an application. Each processor in our target MPSoC system can run more than one task at the same time. It means that multiple tasks (scheduled by FCFS policy) can be mapped onto the same processor. As each tile has the same architecture in our target MPSoC system, compared with the approach used in [25]. We can further simplify the design-time mapping optimising problem

[2] The number of possible mappings of a mapping problem where m tasks need to be mapped onto n heterogeneous processing elements is n^m.

by considering only a partial target architecture (i.e., a tile) to limit the number of processing elements in the mapping problem. The problem of how to further optimise the entire mapping for the target workload scenario and the target MPSoC system will be solved at run time by light-weight heuristics. The details of our proposed task mapping approach will be explained in the following subsections.

4.1.1 Design-time mapping optimisation

At design time, a performance-optimized task mapping (and, if needed, also a power-optimized mapping) for each execution mode of each *application in isolation* is determined by using state-of-the-art scenario-aware Design Space Exploration (DSE) techniques [26,40]. Note that the target architecture in this optimisation process is the tile architecture inside the target MPSoC system as mentioned above. This greatly reduces the complexity of each single task mapping problem. By using this approach, the time needed for finding the pre-optimised task mappings and the memory usage for storing these mappings on the system have been significantly reduced. For example, when considering n target applications with each m execution modes, the number of mappings that need to be optimized and stored is $m * n$ in our case. This number is greatly reduced compared to the $(m + 1)^n - 1$ mappings that need to be optimized and stored in the case of performing mapping preparation for complete workload scenarios. Moreover, if a new application needs to be supported on the target MPSoC system, this would only require providing the pre-optimized mappings of this new application to our SHARA without redoing the entire process of design-time mapping preparation for all possible (new) workload scenarios. Also there is no need to redo the design time mapping optimisation with the architecture scaling of the target MPSoC system, i.e. a larger number of tiles in the target MPSoC.

In each task mapping problem, the communication channels in each application are automatically mapped onto fastest available communication medium in the architecture based on the mapping of two communicating tasks. For example, if two communicating tasks are mapping onto the same processing elements, then the communication channel between these two tasks will be mapped onto the inner buffer of the processing element automatically. Figure 7 shows the mapping of an application on the tile architecture we considered in our MPSoC system. At design time, the mappings that need to be explored are expressed as what is shown in the mid-part of Fig. 7. These mappings will be stored in the local memory of the GM. Besides the performance optimised mapping for each execution mode of each isolated application, the execution time of each task on each processor in a tile, the communication time between tasks on different communication channels of the target system and the migrating data size between processors for each task should also be analysed at design time and stored on the target system for mapping optimisation and adaptivity throttling.

4.1.2 Tile-level workload partition

As the mappings prepared at design time are optimised targeting the hardware resources inside a tile. To fully utilise the resources of our target MPSoC system where multiple identical tiles are present, the application level parallelism in a workload scenario will be addressed in the GM by means of workload partition. When the workload scenario on the target system changes, the mapping of applications (application to tile) in this new scenario may be adjusted by using a load balancing heuristic as shown in Algorithm 1. It means that the workload partition is triggered by the change of the workload scenario on the target

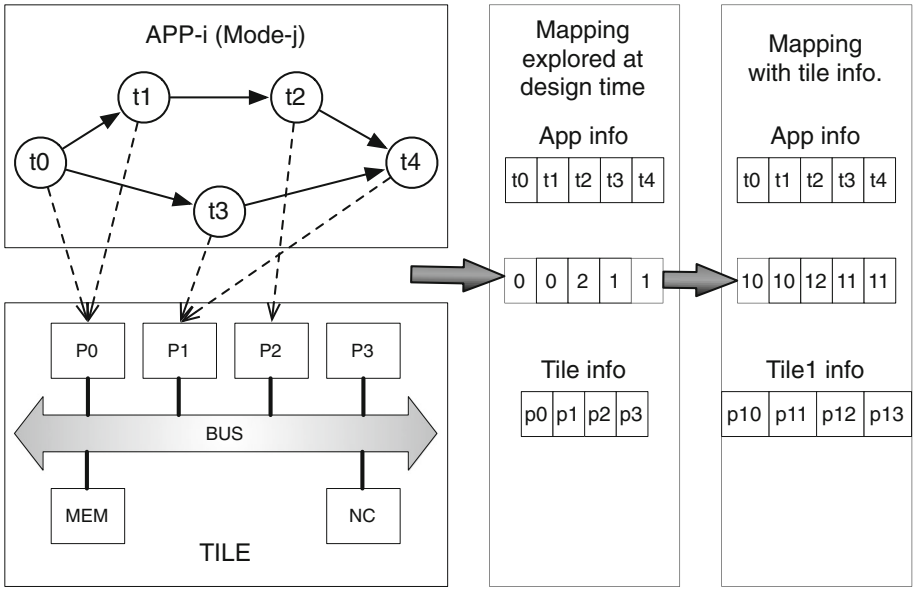

Fig. 7 A simple example of encoded task mapping

Algorithm 1: The heuristic of workload partition in the GM

Input: (μ_{old},η_{old}), $MPSoC$
Output: (μ_{new},η_{new})
1: $(\mu_{new},\eta_{new}) = = (\mu_{old},\eta_{old})$;
2: U = tileUsage((μ_{new},η_{new}), $MPSoC$);
3: $U_{max'}$ = max(U); $U_{max} = +\infty$;
4: $U_{min'} = U_{min}$ = min(U);
5: while $U_{max'} < U_{max}$:
6: $U_{max} = U_{max'}$
7: $U_{min} = U_{min'}$
8: app_{umin} = getMinApp(tile with U_{max})
9: (μ^*,η^*) = getOptMapping(app_{umin});
10: (μ_{app},η_{app}) = app2Tile((μ^*,η^*), tile with U_{min});
11: (μ_{new},η_{new}) = change((μ_{app},η_{app}), (μ_{new},η_{new}));
12: U = tileUsage((μ_{new},η_{new}), $MPSoC$);
13: $U_{max'}$ = max(U);
14: $U_{min'}$ = min(U);
* recover to the mapping that satisfies the condition in line 5 *\
15: recover2Prev((μ_{new},η_{new}));
16: return (μ_{new},η_{new});

MPSoC system. For a newly detected workload scenario, the utilisation of each tile will be calculated using the application/system information in the function of line 2 based on the current/old mapping on the system. The actual workload partition process starts from line 5 to line 14 in Algorithm 1. In each iteration of this process, if the maximal resource usage among tiles can be reduced, the application with smallest resource consumption (line 8) on the tile with maximal resource utilisation will be reallocated onto the tile with minimal resource utilisation. It means that the algorithm tries to gradually balance the system by migrating applications from overloaded tiles to lightly-loaded tiles. When an application is reallocated

to a different tile, its pre-optimised mapping will be used on the new tile as shown in line 9 of Algorithm 1. This process will continue until the workloads on the system are well balanced. As the task migration overhead will greatly influence the system performance as can be seen in the experiment section, this algorithm tries to balance the system workload with a minimal number of task migrations among tiles to reduce the tile-level task migration overhead.

4.1.3 Processor-level task mapping optimisation

After the entire new workload scenario is reallocated by the GM, each tile on the target MPSoC system might need to execute a new tile-level scenario. In the workload partition process, it only focuses on the total resource consumption of a complete application. The task mapping of an application on the resources inside a tile is either generated from the pre-optimised mappings stored on the system or the mapping preserved from the previous workload scenario. However, simply merging per-application mappings might not be good enough with regard to the optimising goal like the performance objective considered in this work. After the GM finished the workload scheduling, the LM in the tile where the new workload has to be executed will further optimise the task mapping derived from the workload partition heuristic in the GM. In each LM, we adopt the EIM algorithm proposed in [25] without considering the energy constraint which generates mappings with good quality in system throughput to further optimise the mapping for the new workload. The pseudocode of this algorithm is outlined in Algorithm 2. It benefits from both the high mapping quality of the design-time static task mapping approaches and the efficiency of the run-time dynamic task mapping approaches. Under this heuristic, for a workload with only a single application active, the EIM algorithm directly outputs the corresponding pre-optimised mapping (the mapping for the particular execution mode of the active application) stored on the system memory as the final mapping. However, for a workload scenario where multiple applications are active simultaneously, it will further iteratively optimise the merged mapping derived in the GM by changing the mapping of tasks on overloaded processors to light loaded processors. It is a greedy algorithm and aims at improving the mapping performance by minimizing the maximal processor usage and processor usage variation among processors in a tile. The algorithm iteratively reduces these two objectives by unloading a task or task bundles from the most heavy loaded processor (under the new mapping) to other processors.

Figure 8 shows a simple example of mapping a workload scenario onto our target MPSoC system using our proposed task mapping approach. In this example, a workload scenario

Algorithm 2: Modefied EIM algorithm

Input: $KPN_{app_{active}}$, $MPSoC$, scenario_id(s_i)
Output: (μ, η) /*new mapping
1: (μ, η) = getInitMapping(s_i); /* merge the optimised mapping of each application*/
2: if singleAppActive(s_i) == true:
3: return (μ, η);
4: else:
5: U = peUsage($KPN_{app_{active}}$, $MPSoC$, μ, η);
6: M_p = maxPUsage(U); /* maximal usage among processors*/
7: V_p = varPUsage(U); /* processor usage variation*/
8: if M_p or V_p is reduced
9: goto line5
10: else
11: return (μ, η)

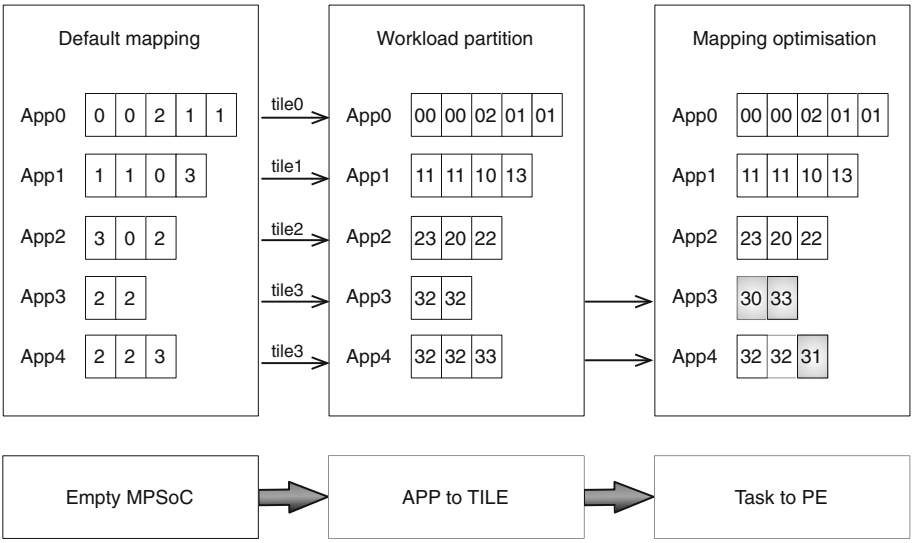

Fig. 8 A simple example of mapping a workload scenario on the target MPSoC system

with five applications is mapped on the empty tile-based MPSoC system. When the GM in SHARA detects the new workload scenario, it will allocate these new applications onto tiles available in the target system by the tile-level workload partition algorithm. After that, each LM starts to further optimise the mapping of applications that are allocated on the corresponding tile. As only one application is active on $tile0$ to $tile2$, the LM on these three tiles will not further optimise the default mapping as it has already been optimised at design time. However, the mapping of applications on $tile3$ is further optimised by the EIM algorithm to improve the mapping quality. This hierarchical task mapping approach of our SHARA framework can greatly simplify the computational complexity comparing to the exhaustive mapping exploration approach not only on the number of mappings need to be explored for all target possible application scenarios as mentioned in Sect. 4.1.1 but also on the complexity of computing each mapping solution. For each single mapping solution, the complexity of the exhaustive approach depends on both the number of tasks in the target application scenario and the number of processors in a tile of our target MPSoC system (under the assumption that the tasks of an application can only be mapped onto a single tile). However, our mapping exploration approach only needs to consider the number of tasks of each application in isolation and the number of processors in a tile. With regard to the quality of mappings derived from our approach and the statically optimised mapping of the exhaustive approach (with an acceptable mapping exploration time like a few hours), the mapping quality of our approach is very close to the later one (under 10 % performance loss in average for our tested application scenarios in the experiment section).

4.2 Adaptivity Throttling for System Reconfiguration

In current scenario-aware adaptive MPSoCs, the system will typically be reconfigured (i.e., the task mapping will be adapted) when a new workload scenario appears on the system, irrespective of the trade-off between reconfiguration costs and benefits. This implies that task migration might occur during this reconfiguration process of which its cost cannot be

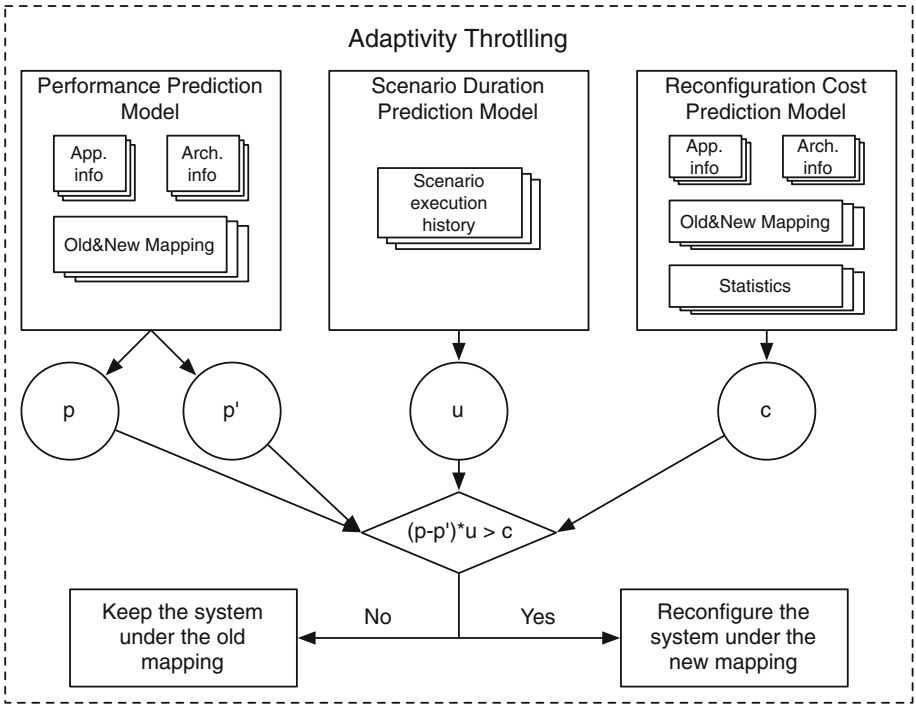

Fig. 9 Adaptivity throttling mechanism in SHARA

ignored, especially for heterogeneous MPSoC systems and those cases where the duration of workload scenarios are relatively short.

Let's assume that the new workload scenario needs to finish u units of work. Under the current/old task mapping, the target MPSoC system requires p units of time to finish a single unit of this work. However, if this new scenario is executing under the new mapping optimised at run time, the execution time of a single unit work of this scenario will be reduced to p'. Consequently, for the target u units of work, the execution time saved by reallocating the system resources under the new mapping before execution for this scenario is $\Delta p = (p - p') \times n$. However, this time reduction comes at the cost of task migration between processors and the computational overhead system manager like the SHARA itself (i.e., the time needed for determining the new mapping and deriving the reconfiguration decision). Here we assume this time cost is c. Under these assumptions, it is evident that only if Δp is larger than c—implying that the system actually benefits from the reconfiguration—then the system should re-map the application tasks to improve system efficiency, and otherwise not. This can be seen as *throttling* the adaptivity. Although a similar trade-off for costs and benefits of reconfiguration can be made in terms of power consumption, we will focus on performance in the remainder of the discussion. To make the adaptivity support in MPSoC systems more effective, the resource scheduler should be capable of explicitly making these reconfiguration decisions (i.e., provide support for adaptivity throttling) whenever workload scenarios change.

Figure 9 give an overview of the adaptivity throttling implementation in the SHARA framework. It is used for adaptively determining system reconfiguration decisions when workload scenario changes. To determine a reconfiguration decision, three parameters are

required: the performance improvement of re-mapping tasks ($p - p'$), the scenario execution duration (u), and the reconfiguration cost (c). These three parameters are, however, unknown before the system reconfiguration. As a consequence, prediction models should be used to predict each of these values. Figure 9 illustrates how to derive a suitable reconfiguration decision based on the outcome of the prediction models (i.e, $\Delta p > c$). In this figure, the information about the target applications and hardware architecture used in the performance prediction model as well as the reconfiguration cost prediction model are prepared at design time as mentioned in Sect. 4.1.1.

The prediction models used for adaptivity throttling cannot be computationally intensive as they have to efficiently make a reconfiguration decision at run time. For the performance and reconfiguration cost prediction, simple analytical models are used in our SHARA framework. However, different with these two parameters, the scenario execution duration is a dynamic parameter which could be heavily influenced by user behaviour or system execution environment. It can't be predicted simply by analytical models. For the prediction of such kind of parameters, a history-based predictor is commonly used which predicts the future behaviour based on the history information. The details of these predictors in our SHARA framework are introduced as follows.

4.2.1 Mapping performance prediction

Our target MPSoC system consists of several homogeneous tiles. Each tile has a typical heterogeneous MPSoC architecture. For this kind of system, we can apply the same performance analytic model to the different tiles in our target MPSoC. The LM of each tile has an instance of the performance model. In this work, we use a simple linear analytic model to predict the performance of different task mappings for workload scenarios targeting our tile architecture.

As our target applications belong to the domain of streaming applications that continuously process an incoming stream of data elements. To capture the duration (u) of a workload scenario in this case, we use the concept of *scenario frames* to define the workload of active applications. Here, we define one *scenario frame* as the time it takes for each active application within a specific workload scenario to process at least a single unit (frame) of data (e.g., processing a single MP3 frame, an H264 frame, etc.). Consequently, the performance (in scenario execution time for one scenario frame) of a workload scenario s_i under mapping tm_i^j can be calculated by Eq. 1

$$p_i^j = max\left(p_{ij}^k\right) \tag{1}$$

where p_{ij}^k represents the frame execution time of app_k which is active in s_i under mapping tm_i^j.

According to the pre-stored application/architecture information, the performance of each active application p_{ij}^k is predicted by Eq. 2.

$$p_{ij}^k = CC_{ij}^k + BK_{ij}^k \tag{2a}$$

$$CC_{ij}^k = \sum_{0 \le m < t} et_{ij}^{km} + \sum_{0 \le n < l} ec_{ij}^{kn} \tag{2b}$$

$$BK_{ij}^k = \sum_{q \neq k} \sum_{t_i^{qs} \in T_{ij}^{qk}} et_{ij}^{qs} + \sum_{q \neq k} \sum_{c_i^{qs} \in C_{ij}^{qk}} ec_{ij}^{qs} \tag{2c}$$

where CC_{ij}^{k} represents a conservative estimate (no concurrency is taken into account) of the total execution time of app_k in scenario s_i under mapping tm_i^j of all t tasks and l communications in app_k. BK_{ij}^{k} is the total time of tasks $t_i^{qs} \in T_{ij}^{qk}$ and communications $c_i^{qs} \in C_{ij}^{qk}$ from other active applications different with app_k. Here, T_{ij}^{qk} and C_{ij}^{qk} is the set of tasks and communications from app_q that have the same mapping (under tm_i^j) with any task and communication of app_k respectively.

4.3 Reconfiguration cost prediction

The reconfiguration cost on our target MPSoC system includes two part: the computational overhead in the GM/LM of our SHARA framework and the task migration cost during system reconfiguration. The overhead of GM/LM is determined by means of measurements and the task migration cost is calculated by Eq. 3b. The model of task migration cost prediction is a simple linear analytic model. The reason behind this model is based on the task migration mechanism we implemented on the target MPSoC system where a *task recreation* [28] mechanism is considered in our system. During task migration, the migrating task will be killed on the original processor and the task state information (and task binary code if needed) will be transferred to the destination processor. The destination processor will load the binary code and state information to restore the task. Here, we label the computational overhead in SHARA as $Comp$ and the task migration cost as $CMig$. Consequently, for a certain workload scenario s_i where the old detected task mapping is tm_i^j and the new generated mapping is $tm_i^{j'}$, it's reconfiguration cost can be derived by the following equation. Notice that, in our MPSoC system, there are two levels of system reconfiguration: the tile-level system reconfiguration and processor-level system reconfiguration. Consequently, to calculate the cost of tile-level system reconfiguration, the parameter $Comp$ and $CMig$ represents the computational overhead in the GM and the inter-tile task migration respectively. For processor-level system reconfiguration cost prediction, they are the computational overhead of the LM and the intra-tile task migration in the corresponding tile. This model is used in both the GM and each LM as a reconfiguration decision has to be predicted at both the inter- and intra-tile level.

$$c_i^{jj'} = Comp_i^{jj'} + CMig_i^{jj'} \tag{3a}$$

$$CMig_i^{jj'} = \left(\sum_{t_k \in T_i^{jj'}} ms_k \right) \Big/ r_{comm} \tag{3b}$$

where $T_i^{jj'}$ is the set of tasks in workload scenario s_i that need to be migrated from mapping tm_i^j to $tm_i^{j'}$, ms_k represents the amount of migrating data for task t_k, and r_{comm} is the speed communication channel used for data transferring on the system.

4.4 Scenario duration prediction

For the purpose of scenario duration prediction, we use the scenario execution history information to predict the future execution behaviour of workload scenarios. Actually, the accuracy of this predictor highly depends on the target application domain. For example, in a periodic system, the scenario execution behaviour is easy to predict by typical history-based predictors such as last value predictor, table-based predictor and the Statistical Metric Model (SMM)

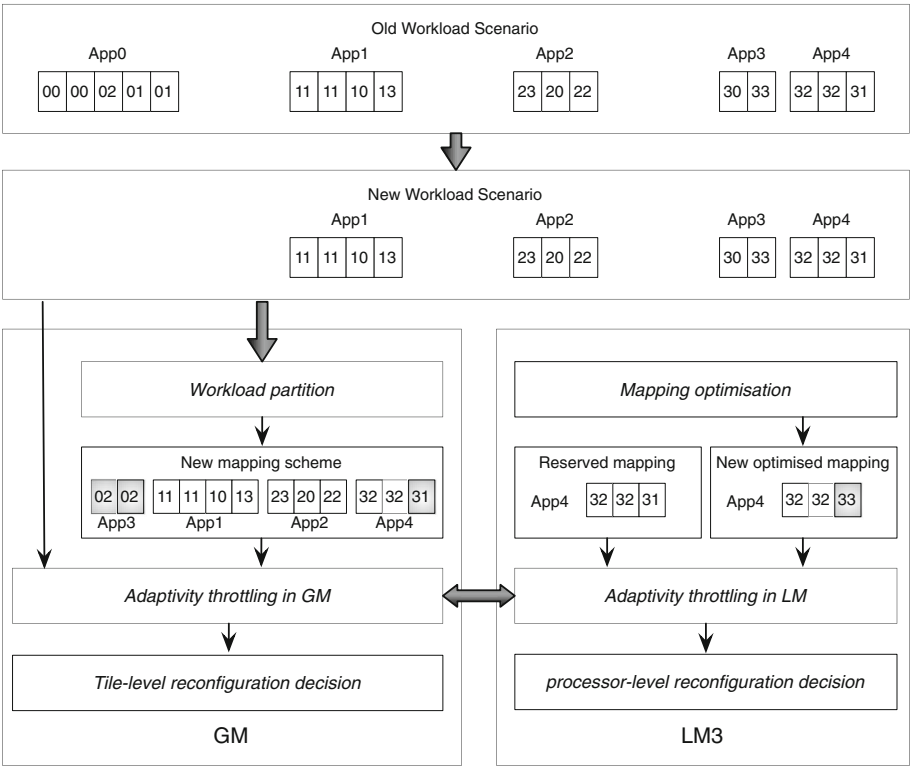

Fig. 10 Hierarchical adaptivity throttling in SHARA

[30]. However, in our multi-media application domain, the workload scenarios usually has a random execution duration behaviour. It is hard to use these predictors to accurately predict the scenario execution duration. Further more, these predictors especially the table-based predictor and the SMM need to consume a certain amount of system memory at run time to record the history information for each workload scenario. In this work, we assume a large number workload scenarios will be executed on the target system. If such predictors are adopted, the memory usage will be a big concern on our MPSoC system. Consequently, we only use the average scenario duration of previous executions of a workload scenario as the future execution duration value. And this average scenario duration information will be updated by the GM after the workload scenario actually finished its execution. This simple scenario duration predictor is initialised in the GM. When a new workload scenario is detected, the GM will predict a scenario duration for tile-level adaptivity throttling and send this predicted value to each LM for processor-level adaptivity throttling.

4.5 Hierarchical adaptivity throttling in SHARA

Using the introduced adaptivity throttling mechanism, a hierarchical scheduling policy is implemented in the SHARA framework where the GM actually schedules the system resources at tile level for new workload scenarios based on the tile-level reconfiguration decision and each LM schedules the resources inside the tile according to processor-level reconfiguration decision. Figure 10 shows how this hierarchical adaptivity throttling approach

works in our SHARA framework. To derive a tile-level reconfiguration decision using the adaptivity throttling mechanism in the GM, those parameters needed for predicting the reconfiguration benefits should target the whole complete workload scenario on the target system. The performance improvement prediction in the GM happens after a new mapping is generated by the workload partition algorithm. The GM will firstly send the corresponding mapping information to the LM in each tile. After each LM predicted the performance in that tile, it will send back the performance information to the GM. The system performance of a whole workload scenario is then determined by the GM using Eq. 2. The tile-level reconfiguration cost depends on both the computational overhead in the GM and the task migration cost between tiles via the NoC. In the example of Fig. 10, the possible task migration cost concerns migrating application $App3$ from $tile3$ to $tile0$. In each LM, the performance prediction only focuses on the workloads that are allocated to the tile by the GM and the reconfiguration cost includes the computational overhead in itself and the possible task migrations (like the third task of $App4$) inside a tile via the local shared bus. With regard to the scenario execution duration prediction, the GM and each LM use the same prediction.

In this work, we limit the number of processors in our target architecture to demonstrate the effectiveness of our proposed approach. The number of processors can be increased as long as the architecture is under our assumption (processors can be divided into identical tiles). For example, we can increase the number of processors in a tile and also the number of tiles in the architecture without any change to our framework. However, if a large number of totally different processors need to be considered, the hybrid task mapping approach in this work will not work well anymore. As it is timing unacceptable to explore optimal mappings for applications under hundreds of processors at design time. Consequently, the tile-level workload partition strategy and the processor-level task mapping algorithm should be adjusted accordingly. For the tile-level workload partition, new approaches should be considered to evaluate the resource usage of an application to balance the utilisation of tiles. With regard to the processor-level task mapping algorithms, on-the-fly run-time task mapping approaches like Bin Packing algorithms can be considered to derive a mapping in a short time without using statically optimised mappings. For adaptivity throttling, a new mapping performance predictor like the worst-case performance evaluation can be considered before an actual mapping is derived by the mapping algorithm.

5 Experiments

5.1 Experiment setup

To illustrate the effectivity of our SHARA framework, we deploy the system-level MPSoC simulation framework from the work of [28] which is based on the open source Sesame simulator [24]. This Sesame-based modeling and simulation environment facilitates efficient performance analysis of embedded (media) systems architectures. The most important feature of this simulator for this work is its ability to support the simulation of run-time system reconfiguration of MPSoC systems. This makes the modeling and simulation of our SHARA instance in this simulator relatively easy. In our experiments, we adapt a dedicated implementation of our SHARA framework on the target architecture model. It means that the GM and the LMs are separately integrated into the system by dedicated hardwares where the GM is connected to each LM via dedicated channel and each LM located in each tile can control the hardware resources inside a tile through dedicated control channels.

In our experiments, we would like to show how our SHARA framework improves the system performance by applying the hierarchical task mapping and adaptivity throttling. The actual functionality of the applications is not very important for this purpose. Therefore, synthetic streaming applications are considered to simplify the simulation process. We use 16 synthetic streaming applications with each application containing only 1 execution mode. In this case, the total number of workload scenarios is 65535 (2^{16}–1). The number of tasks in each application ranges from 4 to 8. We assume that each task can be executed on each processor of the target MPSoC using the corresponding pre-compiled code. The task execution time and migration data size of each task on each processor have been randomly generated and range between 1000 and 100,000 time units (simulation cycles) and between 5K and 50K Bytes respectively. Communications between tasks range from 100 to 10,000 Bytes in size. In our experiments, we assume that all target applications are firstly loaded onto the same tile ($tile0$) with their pre-optimised mappings as an initial state of the system. For generating test scenario sequences in our experiments, we developed a scenario generator in which some parameters like the number of scenarios, the execution duration of scenarios (e.g. [1100] frames) are used for generating random scenario sequences.

5.2 Experimental results

As introduced in the Sect. 4.1.2, the algorithm used in our GM (we refer to it as SC in this experiment) for tile-level workload partition tries to balance the workload among tiles with minimal inter-tile task migration. In the first experiment, we compare it with a load balancing algorithm [27] (noted as BF) to show the effect of our algorithm for reducing the task migration cost while achieving a well balanced system at tile level. In this BF algorithm, the active applications of the new workload scenario will be sorted in resource consumption descending order. And then the applications under this descending order will be gradually allocated to the tile with minimal resource utilisation (the resource utilisation of tiles will be recalculated after each allocation) under the pre-optimised mapping. In this experiment, we do not consider a further mapping optimisation in the LM of each tile and also the adaptivity throttling ability of our SHARA has been deactivated. It means that the system will only optimise the mapping of a newly detected workload scenario by workload partition in the GM. After that, the system will be reconfigured based on the new mapping to execute the new workload scenario. We randomly generate 10,000 workload scenarios and each workload scenario only execute for 1 $scenario frame$ as a scenario sequence. Figure 11

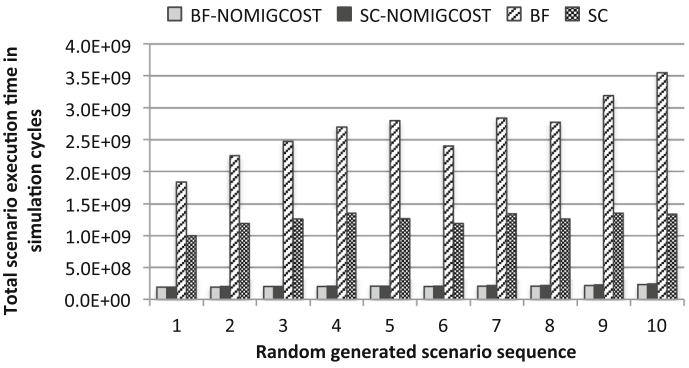

Fig. 11 Comparison of two load balancing algorithms for tile-level workload partition

gives the results of executing 10 such scenario sequences under these two algorithms where *BF-NOMIGCOST* and *SC-NOMIGCOST* represent the results without considering the tile-level system reconfiguration cost by using BF and SC respectively. From the results, we can clearly see that our algorithm has similar performance compared with BF if we ignore the reconfiguration cost. However, when the cost of system reconfiguration is taken into consideration, our algorithm performs much better than BF as the BF algorithm does not take the previous position of each application into account for workload distribution. In this experiment, the task migration cost comes from migrating an application from one tile to another tile. This tile-level task migration overhead is very heavy as slow communication channels between tiles will be used for task migrations. To reduce the tile-level task migration overhead, the workload partition algorithm in the GM should keep the number of tile-level task migration as low as possible.

After investigating the mapping quality and tile-level reconfiguration cost by applying the global task mapping optimisation in the GM, we further study the hierarchical mapping optimisation approach of our SHARA framework in the second experiment. In this experiment, we still ignore the adaptivity throttling ability of our framework (task migration happens when the newly derived mapping is different with the old mapping). We select two workload scenarios $S16$ and $S4$ as our target scenarios to show how the scenario execution time is influenced by system reconfiguration. The scenario $S16$, in which all the target applications are active, is the most complex workload scenario of all possible scenarios. And $S4$ is a scenario where only four applications are active. In this experiment, our hierarchical mapping optimisation approach (as it contains two steps of mapping optimisation in the GM and LMs, here we label it as $GM-LM$) is compared to three other approaches $NGM-NLM$, $NGM-LM$ and $GM-NLM$. The $NGM-NLM$ approach does not contain any mapping optimisation process. It means that, in this approach, the mapping in the initial state of the target system is directly used for executing the target two workload scenarios. The $GM-NLM$ and $NGM-LM$ only considers the tile-level and process-level mapping optimisation for the target scenarios from the initial system state respectively. In this experiment, we assume that the system will be triggered for reconfiguration when the first workload scenario is detected on the system. For the target two workload scenarios $S16$ and $S4$, they are separately executed for a single *scenario frame* directly from the initial system state (all target applications are loaded onto $tile0$). The results of this experiment are illustrated in Fig. 12. In this figure, the x-axis represents different states of the two target scenarios where for example *S16-NOCOST* and $S16-COST$ are executing the scenario $S16$ without and with considering all system reconfiguration cost

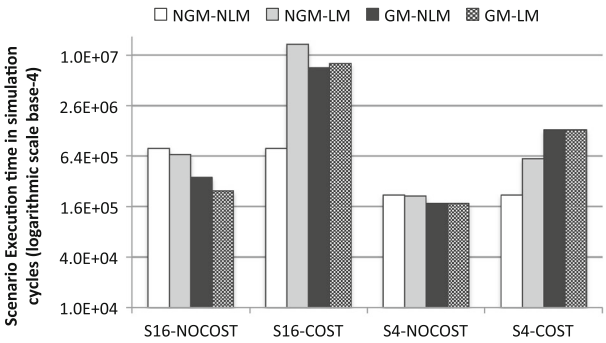

Fig. 12 Performance comparison of different task optimisation approaches

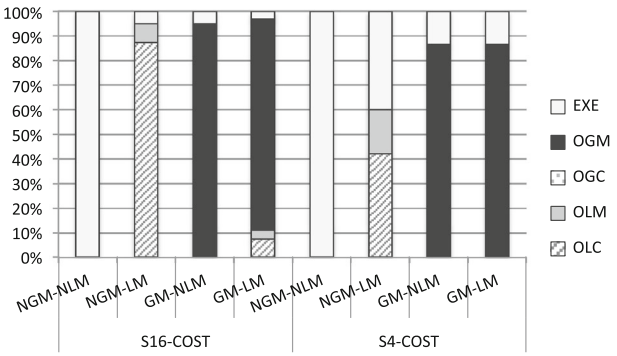

Fig. 13 System reconfiguration cost in $S16-COST$ and $S4-COST$

(both tile-level and process-level) respectively. Here, the results of $*-NOCOST$ is derived by directly executing the corresponding scenario for a single frame under the mapping optimised by the various approaches.

If we only consider the quality of the mapping ($*-NOCOST$) derived from different approaches, from the experimental results, we notice that the mapping optimisation in the GM is more important compared with the optimisation in the LMs. Compared to the approach of $NGM-NLM$, the other three approaches $NGM-LM$, $GM-NLM$ and $GM-LM$ improve the scenario performance by 17, 121 and 220 % respectively in $S16-NOCOST$ and 3, 26 and 26 % in $S4-NOCOST$. In the complex scenario case $S16$, the GM and LMs are able to greatly improve the mapping quality. However, when the scenario is relatively simple like $S4$ where the resource contention is not critical, the performance improvement is not that apparent anymore especially the improvement from the optimisation by the LMs. When taking the system reconfiguration cost into consideration, we can see from the results shown in Fig. 12 that the system reconfiguration cost which contains both the task migration cost and the computational overhead in the GM and LMs will dominate the execution time of scenarios if the scenario duration (number of $scenario frames$) is very short. In our test cases, as we set the execution duration of each scenario to one $scenario frame$, consequently the final performance (reconfiguration cost included) of $NGM-NLM$ is much better than the other three approaches. To further understand where the system reconfiguration cost comes from, we zoom into the scenario execution time of $S16-COST$ and $S4-COST$ in Fig. 13. In this figure, the symbols of EXE, OGM, OGC, OLM and OLC respectively represent the actual execution time of the target scenario under the mapping optimised by the corresponding approach, the overhead of inter-tile (global) task migration, the computational overhead of the GM, the overhead of intra-tile (local) task migration and the computational overhead of LMs. Note that, as the LMs in our system work in parallel, the OLM (the overhead of intra-tile task migration) and the OLC (the computational overhead of LM) come from the tile where the intra-tile task migration cost and the computational overhead in the LM in total is the maximal among tiles. From Fig. 13, we clearly see that when the GM takes part in the mapping optimisation process, the system reconfiguration cost mainly comes from the task migration between tiles. Considering the processor-level system reconfiguration, the overhead is dominated by the computational cost in LMs especially when the number of tasks that are allocated to the tile is very large. The reason behind that can be explained as follows. In our experiment, the mapping used for further optimisation in each tile is merged from the pre-optimised mapping of each application. This original mapping normally is already

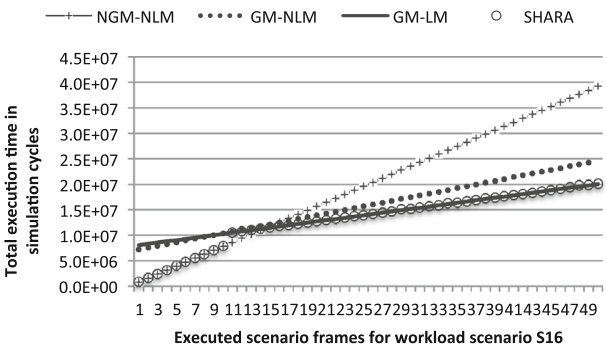

Fig. 14 The influence of scenario duration to final system performance under different approaches

well balanced among processors in each tile. The algorithm used in each LM will take a relatively large time to further improve the mapping quality with only a few tasks that need to be migrated among processors.

From the second experiment, we can see that if the scenario execution duration is very short, the system should not be reconfigured as the large system reconfiguration cost will neutralize the performance improvement by run-time task mapping optimisation. Consequently, in the third experiment, we would like to show how the system performance is influenced by the scenario execution duration in our target large-scale MPSoC system. For this purpose, we investigate the workload scenario $S16$ of the second experiment with a gradually increasing scenario execution duration. In this experiment, we use the complete SHARA framework (the adaptivity throttling is enabled) for run-time resource allocation and compare it with the three approaches $NGM-NLM$, $GM-NLM$ and $GM-LM$ considered in the second experiment. Figure 14 shows the total execution time including the system reconfiguration cost of different scenario durations under different resource allocation approaches. Clearly, as the $NGM-NLM$ does not need application remapping, the total execution time increases linearly with the scenario duration (in $scenario frames$). Similar behaviour can be found in $GM-NLM$ and $GM-LM$. However, as the system is reconfigured according to the corresponding optimised mapping at the beginning of the scenario $S16$, the total execution time has a slower increase with the scenario duration under these two approaches compared to $NGM-NLM$. As the mapping quality derived by $GM-LM$ is better than the one derived by $GM-NLM$, the total execution time of the former approach has an even slower increase with scenario duration. From these three approaches, we can see that the $NGM-NLM$ has the best performance when the scenario duration is small (for example, under 15 scenario frames in our test case) as it avoids the system reconfiguration cost. However, with the increase of scenario execution duration, it is increasingly outperformed by $GM-NLM$ and $GM-LM$.

By using the adaptivity throttling ability of our SHARA framework, we are able to solve the drawback of the other approaches. When the scenario execution duration is small, the system will be kept unchanged to avoid unnecessary system reconfigurations. On the other hand, when the scenario execution duration is large, the system will be reconfigured to the mapping optimised by SHARA. In this experiment, the scenario duration predictor has been deactivated[3] to exclude its influence for deriving a reconfiguration decision which will be further studied in the next experiment. The results of using our complete SHARA framework shown in Fig. 14 verify the ability of improving the system performance by our hierarchical

[3] We directly use the actual execution duration of the target workload scenario for reconfiguration prediction.

adaptivity throttling approach on the target large-scale MPSoC system. When the scenario duration of scenario $S16$ is lower than 11, *SHARA* is very close to $NGM-NLM$. After that, it is very close to $GM-LM$. This also reflects the fact that the overhead of adaptivity throttling is small enough to be ignored. However, notice that, from 11 to 15 in the x-axis of Fig. 14, the results of *SHARA* are close to $GM-LM$. If the prediction models used for adaptivity throttling in SHARA are absolutely accurate, these points should have been close to $NGM-NLM$. The prediction models used for adaptivity throttling in our framework are relatively simple first-order models. For the performance prediction model, the prediction error is 15.5 % in average for our target workload scenarios. With regard to the reconfiguration cost prediction model, there is no prediction error in our experiments as our simulator directly use the task migration cost model in Sect. 4.3 and the run-time computational overhead of our framework is directly mearsured. Currently, we do not have a hardware platform to implement our framework, the task migration cost prediction error can not be mearsured. For the scenario duration prediction, the average scenario duration is adopted in our framework. This approach has a good performance for scenarios with a stable execution behavior. However, when the execution duration of a scenario varies greatly, the prediction error will be very large. And this will greatly influence the performance of the adaptivity throttling. Considering all the factors together, in this specific experiment, there is 8 % prediction error for whether the system needs to be reconfigured.

In the fourth experiment, we apply our proposed SHARA framework in more complex scenario cases on the target MPSoC system to test its performance when scenario duration prediction is considered and compare the results with two approaches *STATIC* and again *GMLM*. In the *STATIC* approach, all applications are statically mapped (i.e., no run-time mapping takes place) using a mapping which has shown to be optimal *on average* for all possible workload scenarios. The *GMLM* is similar to a normally used approach in small scale MPSoCs where the system will always be reconfigured based on the optimised mapping when a new workload scenario is detected. To model dynamic application behaviour over time (e.g. due to user behaviour), we generate three kinds of workload scenario sequences. Each sequence is generated in two steps. The first step is to randomly choose a workload scenario from all the possible workload scenarios. For each selected workload scenario, it will appear in the scenario sequence for multiple times. The second step is to generate the duration in *scenario frames* for each appearance of the selected workload scenario. In this experiment, we model totally random user behaviour to show the performance of our approach in extreme cases. For this purpose, the scenario duration is generated by a random generator with a certain average scenario duration (number of $scenario frames$). This process iterates until a pre-defined total frame number (10,000 frames in our case) has been achieved for each scenario sequence. Our three target scenario sequences $seq10$, $seq100$ and $seq10-100$ in Fig. 15 are distinguished by the average number of scenario frames set for the random scenario duration generator where an average frame number of 10, 100 and 10 to 100 (the average frame number set for the generator in each iteration is also randomly derived from 10 to 100) are used for generating the three kinds of sequence respectively.

The results of each kind of scenario sequence shown in Fig. 15 are averaged over five randomly generated different sequences. From this figure, we can see that our *SHARA* approach has a good trade off between *STATIC* and *GMLM*. When the average number of scenario frames for each workload scenario is small like $seq10$, the *GMLM* approach where a system reconfiguration alway happens when a new scenario is detected has the worst performance. However, it has the best performance when the average scenario frame is large like $seq100$ as in this case the system reconfiguration cost is covered by the performance improvement because of system reconfiguration. If the average number of scenario frames becomes even

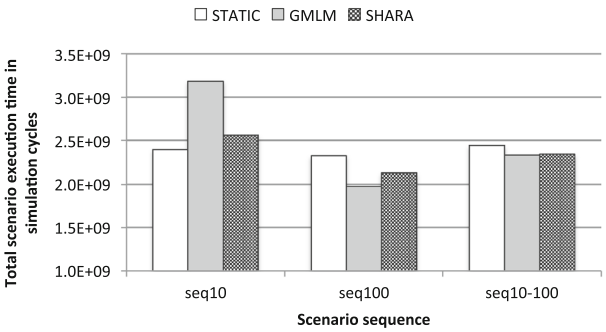

Fig. 15 Performance of SHARA in more complex scenario cases

larger, the gap between *STATIC* and *GMLM*, *SHARA* will also increase. Comparing *SHARA* with *GMLM*, in the case of $seq100$, our *SHARA* approach suffers from both the reconfiguration prediction overhead and error system reconfiguration prediction which mainly caused by the prediction error in mapping performance and task migration cost as mentioned in the third experiment. In the case of $seq10-100$, the scenario duration predictor also influences the prediction of system reconfiguration. However, as the system performance degradation caused by errors in the system reconfiguration prediction in *SHARA* is almost equal to the unnecessary reconfiguration overhead in *GMLM* when the average scenario duration is short, our *SHARA* shows a similar performance with *GMLM*. The problem of how to improve the system performance by optimising the prediction accuracy of our predictors used for adaptivity throttling will be further studied in future work. Overall, our approach can get a trade off between STATIC and GMLM in different scenario situations. It means that in the case of workload scenarios with small execution duration, our SHARA's performance is close to STATIC (the best solution in this case). On the other hand, for workload scenarios with large execution duration, our SHARA's performance is close to GMLM (the best solution in this case).

In our last experiment, we compare our SHARA resource management approach with a classic light-weight mapping algorithm—First Fit Bin Packing (FFBP) [9] on a system that contains more tiles than the system considered in the previous experiments. The new system has 16 tiles connected by a 2-D mesh NoC similar to the one in Fig. 2. For this system, we duplicate the applications in our previous experiments to 64 applications (each application task has a different execution time on each processor). With these applications, we generate three kinds of scenario sequence $seq1$, $seq100$ and $seq500$ (each scenario only active for 1, 100 and 500 frames in each appearance) using the method in the last experiment.But in this experiment, we considered a total number of 100,000 frames for each scenario sequence. Under these scenario sequences, we compare the performance (total scenario execution time) of our SHARA with FFBP. The results (normalised to FFBP) of this experiment is shown in Fig. 16. From this figure, we can see that our *SHARA* approach has a good performance under $seq100$ and $seq500$ but a bad performance under $seq1$. This is similar to the results shown in the fourth experiment. When the scenario execution duration is large enough to cover the system reconfiguration overhead, our approach can have a good performance. However, as the size of the target system considered in this experiment is large, the system reconfiguration overhead especially the overhead of the global manager is heavy. Therefore, the performance improvement over FFBP is not remarkable. The global manager in our SHARA framework

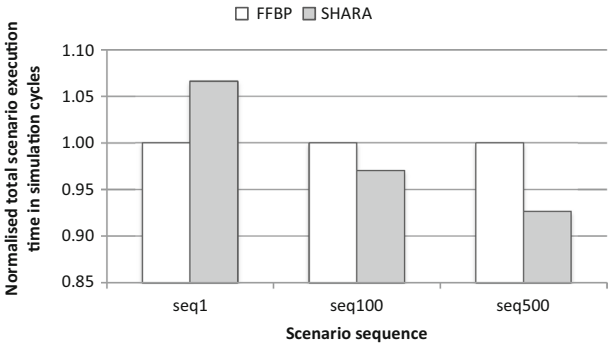

Fig. 16 Performance comparison between SHARA and FFBP on a larger tile-based system

may become a bottleneck as the size of the system scale. This problem will be considered in our future work.

Regarding to the run-time system storage consumption of our SHARA framework, several assumptions should be mentioned. On our target MPSoC system, we store all the design-time prepared information in the local memory of the GM. For storing the pre-optimised mappings, we assume that the mapping information of each task and each communication channel between tasks is stored in one $byte$. In our target synthetic streaming applications, there are 88 tasks and 67 communication channels in total. Consequently, to store the pre-optimised mappings, the memory usage is 155 $bytes$. Beside the pre-optimised mappings, in our *SHARA* framework, we also need to store the application/system information and the average scenario execution history information. Here, we assume that each piece of this information needs one $word$ of memory. Consequently, for storing the application/system information, 1408 $bytes$ of memory are required. With regard to the average scenario execution history information, as each workload scenario needs one $word$ to store the information, the total memory usage is 256 KB. This memory consumption can be reduced by using smaller memory bits to record the execution history information of each workload scenario.

6 Related research

In recent years, much research has been performed for increasing the system adaptivity by using dynamic run-time task remapping to achieve better performance or save energy consumption [34,35,39]. Among these research, the hybrid task mapping approach is commonly used which combines the design-time preparation with the run-time dynamic mapping policy to do task reallocation. For example, Mariani et al. [17] proposed a run-time management framework in which Pareto-fronts with system configuration points for different applications are determined during design-time DSE, after which heuristics are used to dynamically select a proper system configuration at run time. In [45], a fast and light-weight priority based heuristic is used to select near-optimal configurations explored at design time for the active applications according to the available platform resources. Reference [37] proposes DSE strategies that perform exploration in view of optimizing throughput and energy consumption by considering a generic platform. The design points derived from the DSE will be selected efficiently at run time. In [32], Schranzhofer et al. proposed static and dynamic task mapping approaches for probabilistic applications based on static and dynamic power

components. Statically pre-computed template mappings for each execution probability are stored on the system and applied at run time, allowing the system to adapt to changing environment conditions. Based on this work, Ref. [10] presents an extension that considers only the static mapping and takes into account the communication and reconfiguration energy component.

Even though these hybrid task mapping approaches can greatly improve the adaptivity for small scale MPSoC system with only a certain number of scenarios or applications need to be supported. Most of them still lack of scalability when the task mapping problem becomes complex in large-scale system with a huge number applications. To solve this problem, several distributed resource management approach for large-scale MPSoC systems or many-core systems have been proposed like the work in [1,7]. In [7], the authors proposed a new concept - invasive computing - for resource management on a heterogeneous, tile-based manycore system. This invasive computing technique uses a multi-agent management layer underpinned by distributed runtime and OS services to support a flexible resource management. The agent of each application executing in the system tries to increase the speedup of its application by acquiring additional cores from the nearby regions. Reference [1] presents a scheme for run- time application mapping in a distributed manner using agents targeting adaptive NoC-based heterogeneous multi-processor systems. Compared to these approaches, our approach uses a hierarchical resource management approach and explicitly studies the influence of system reconfiguration for run-time resource allocation. Recently, Ref. [31] also proposed a scenario-based run-time mapping approach for many-core systems which is very similar to our work. In their approach, the execution scenarios are combined into a finite state machine and the transitions between scenarios are limited in the pre-determined states. However, we do not have such kind of limitations and consequently more complex run-time situations can be considered in our work.

7 Conclusion

In this article, we proposed a scenario-based hierarchical run-time adaptive resource allocation framework to increase the adaptivity of large-scale heterogeneous MPSoC systems where a large number of scenarios or applications need to be supported. The SHARA framework adopts a hierarchical resource allocation mechanism to reduce the complexity of the task mapping problem at run time. In this framework, the system resources are allocated as tiles which could be either real tiles in a tiled system or virtual tiles virtually divided on a system in a global resource management view. Inside each tile, the actual hardware resources will be local allocated to the workload active on it. By using this framework on a large-scale heterogeneous MPSoC system, it is able to support large number of workload scenarios with each of them running under a near optimal mappings derived by our proposed hybrid task mapping approach. In this hybrid approach, optimal (or near optimal) mappings targeting the optimising goal like system performance in this work of each application will be explored at design time and used at run-time by heuristics for further optimisation. For a new workload scenario, after deriving a new mapping, a self-adaptive scheduling policy (adaptivity throttling) will be applied for actual system reconfiguration based on the scenario execution history behaviour. It is helpful to avoid unnecessary system reconfiguration in the case when the reconfiguration is not beneficial. By using this scheduling approach, the system can adapt its behaviour according to the user behaviour. Experiment results confirm the effectiveness of our SHARA framework. In our future work, we will firstly try to implement

our SHARA framework on a real large-scale MPSoC platform and investigate some real-world multi-media applications to improve the efficiency of our SHARA framework. After that, we will also study the possibility of extending our framework for power management on large-scale MPSoC systems. For this purpose, new hierarchical task mapping approaches for minimizing the system's energy consumption are required. Also the adaptivity throttling machenism should target the trade off between the energy overhead and benefit of a system reconfiguration.

References

1. Al Faruque MA, Krist R, Henkel J (2008) Adam: run-time agent-based distributed application mapping for on-chip communication. In: Proceedings of DAC'08, pp 760–765. doi:10.1145/1391469.1391664
2. Bertozzi S, Acquaviva A, Bertozzi D, Poggiali A (2006) Supporting task migration in multi-processor systems-on-chip: A feasibility study. In: Proceedings of the design, automation and test in Europe, 2006. DATE '06, vol 1, pp 1–6. doi:10.1109/DATE.2006.243952
3. Cannella E, Derin O, Meloni P, Tuveri G, Stefanov T (2012) Adaptivity support for mpsocs based on process migration in polyhedral process networks. VLSI Des 2(2–2):2. doi:10.1155/2012/987209
4. Castrillon J, Tretter A, Leupers R, Ascheid G (2012) Communication-aware mapping of kpn applications onto heterogeneous mpsocs. In: Proceedings of the 49th annual design automation conference, DAC'12, pp 1266–1271. ACM, New York. doi:10.1145/2228360.2228597
5. Compton K, Hauck S (2002) Reconfigurable computing: a survey of systems and software. ACM Comput Surv 34(2):171–210. doi:10.1145/508352.508353
6. Gheorghita SV, Palkovic M, Hamers J, Vandecappelle A, Mamagkakis S, Basten T, Eeckhout L, Corporaal H, Catthoor F, Vandeputte F, Bosschere KD (2009) System-scenario-based design of dynamic embedded systems. ACM Trans Des Autom Electr Syst 14(1):3
7. Henkel J, Herkersdorf A, Bauer L, Wild T, Hubner M, Pujari R, Grudnitsky A, Heisswolf J, Zaib A, Vogel B, Lari V, Kobbe S (2012) Invasive manycore architectures. In: Design automation conference (ASP-DAC), 2012 17th Asia and South Pacific, pp 193–200. doi:10.1109/ASPDAC.2012.6164944
8. Howard J, Dighe S, Hoskote Y, Vangal S, Finan D, Ruhl G, Jenkins D, Wilson H, Borkar N, Schrom G, Pailet F, Jain S, Jacob T, Yada S, Marella S, Salihundam P, Erraguntla V, Konow M, Riepen M, Droege G, Lindemann J, Gries M, Apel T, Henriss K, Lund-Larsen T, Steibl S, Borkar S, De V, Van Der Wijngaart R, Mattson T (2010) A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In: 2010 IEEE International solid-state circuits conference digest of technical papers (ISSCC), pp 108–109. doi:10.1109/ISSCC.2010.5434077
9. Huang J, Raabe A, Buckl C, Knoll A (2011) A workflow for runtime adaptive task allocation on heterogeneous mpsocs. In: Proceedings of DATE'11, pp 1119–1134
10. Hussien A, Eltawil A, Amin R, Martin J (2011) Energy aware task mapping algorithm for heterogeneous mpsoc based architectures. In: 2011 IEEE 29th international conference on computer design (ICCD), pp 449–450. doi:10.1109/ICCD.2011.6081444
11. Javaid H, Parameswaran S (2009) A design flow for application specific heterogeneous pipelined multiprocessor systems. In: Proceedings of the 46th annual design automation conference, DAC'09, pp 250–253. ACM, New York. doi:10.1145/1629911.1629979
12. Kahn G (1974) The semantics of a simple language for parallel programming. In: Information processing, pp 471–475. North Holland, Amsterdam
13. Kobbe S, Bauer L, Lohmann D, Schröder-Preikschat W, Henkel J (2011) Distrm: Distributed resource management for on-chip many-core systems. In: Proceedings of the seventh IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES+ISSS '11, pp 119–128. ACM, New York. doi:10.1145/2039370.2039392
14. Manferdelli J, Govindaraju N, Crall C (2008) Challenges and opportunities in many-core computing. Proc IEEE 96(5):808–815. doi:10.1109/JPROC.2008.917730

15. Manolache S, Eles P, Peng Z (2008) Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints. ACM Trans Embed Comput Syst 7(2):19:1–19:35. doi:10.1145/1331331.1331343
16. Marchesan Almeida G, Sassatelli G, Benoit P, Saint-Jean N, Varyani S, Torres L, Robert M (2009) An adaptive message passing mpsoc framework. Int J Reconfig Comput
17. Mariani G, Avasare P, Vanmeerbeeck G, Ykman-Couvreur C, Palermo G, Silvano C, Zaccaria V (2010) An industrial design space exploration framework for supporting run-time resource management on multi-core systems. In: Proceeedings of DATE'10, pp 196–201
18. Mariani G, Avasare P, Vanmeerbeeck G, Ykman-Couvreur C, Palermo G, Silvano C, Zaccaria V (2010) An industrial design space exploration framework for supporting run-time resource management on multi-core systems. In: Design, automation test in europe conference exhibition (DATE), 2010, pp 196–201. doi:10.1109/DATE.2010.5457211
19. Mattson TG, Riepen M, Lehnig T, Brett P, Haas W, Kennedy P, Howard J, Vangal S, Borkar N, Ruhl G, Dighe S (2010) The 48-core scc processor: The programmer's view. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis, SC '10, pp 1–11. IEEE Computer Society, Washington, DC. doi:10.1109/SC.2010.53
20. Melpignano D, Benini L, Flamand E, Jego B, Lepley T, Haugou G, Clermidy F, Dutoit D (2012) Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications. In: Proceedings of the 49th annual design automation conference, DAC '12, pp 1137–1142. ACM, New York. doi:10.1145/2228360.2228568
21. Nollet V, Avasare P, Eeckhaut H, Verkest D, Corporaal H (2008) Run-time management of a mpsoc containing fpga fabric tiles. IEEE Trans Very Large Scale Integr (VLSI) Syst 16(1):24–33. doi:10.1109/TVLSI.2007.912097
22. Paul JM, Thomas DE, Bobrek A (2006) Scenario-oriented design for single-chip heterogeneous multiprocessors. IEEE Trans VLSI Syst 14(8):868–880. doi:10.1109/TVLSI.2006.878474
23. Pimentel A, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112. doi:10.1109/TC.2006.16
24. Pimentel AD, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112
25. Quan W, Pimentel A (2013) An iterative multi-application mapping algorithm for heterogeneous mpsocs. In: 2013 IEEE 11th symposium on embedded systems for real-time multimedia (ESTIMedia), pp 115–124. doi:10.1109/ESTIMedia.2013.6704510
26. Quan W, Pimentel A (2014) Towards exploring vast mpsoc mapping design spaces using a bias-elitist evolutionary approach. In: 2014 17th Euromicro conference on digital system design (DSD), pp 655–658. doi:10.1109/DSD.2014.46
27. Quan W, Pimentel AD (2013) A scenario-based run-time task mapping algorithm for mpsocs. In: Proceedings of the 50th annual design automation conference, DAC'13, pp 131:1–131:6. ACM, New York. doi:10.1145/2463209.2488895
28. Quan W, Pimentel AD (2014) A system-level simulation framework for evaluating task migration in mpsocs. In: Proceedings of the 2014 international conference on compilers, architecture and synthesis for embedded systems, CASES '14, pp 13:1–13:9. ACM, New York. doi:10.1145/2656106.2656111
29. Quan W, Pimentel AD (2015) A hybrid task mapping algorithm for heterogeneous mpsocs. ACM Trans Embed Comput Syst 14(1):14:1–14:25. doi:10.1145/2680542
30. Sarikaya R, Isci C, Buyuktosunoglu A (2013) Runtime application behavior prediction using a statistical metric model. IEEE Trans Comput 62(3):575–588. doi:10.1109/TC.2012.25
31. Schor L, Bacivarov I, Rai D, Yang H, Kang SH, Thiele L (2012) Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Proceedings of the 2012 international conference on compilers, architectures and synthesis for embedded systems, CASES'12, pp 71–80. ACM, New York. doi:10.1145/2380403.2380422
32. Schranzhofer A, Chen JJ, Thiele L (2010) Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms. IEEE Trans Ind Inform 6(4):692–707. doi:10.1109/TII.2010.2062192
33. Shabbir A, Kumar A, Mesman B, Corporaal H (2011) Distributed resource management for concurrent execution of multimedia applications on mpsoc platforms. In: 2011 international conference on embedded computer systems (SAMOS), pp 132–139. doi:10.1109/SAMOS.2011.6045454
34. Shafique M, Henkel J (2013) Agent-based distributed power management for kilo-core processors. In: 2013 IEEE/ACM International conference on computer-aided design (ICCAD), pp 153–160. doi:10.1109/ICCAD.2013.6691112
35. Shafique M, Vogel B, Henkel J (2013) Self-adaptive hybrid dynamic power management for many-core systems. In: Design, automation test in europe conference exhibition (DATE), 2013, pp 51–56. doi:10.7873/DATE.2013.025

36. Shojaei H, Ghamarian AH, Basten T, Geilen M, Stuijk S, Hoes R (2009) A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for cmp run-time management. In: 46th ACM/IEEE design automation conference, 2009. DAC'09, pp 917–922
37. Singh AK, Kumar A, Srikanthan T (2013) Accelerating throughput-aware runtime mapping for heterogeneous mpsocs. ACM Trans Des Autom Electron Syst 18(1):9:1–9:29. doi:10.1145/2390191.2390200
38. Singh AK, Shafique M, Kumar A, Henkel J (2013) Mapping on multi/many-core systems: survey of current and emerging trends. In: Proceedings of the 50th annual design automation conference, DAC '13, pp 1:1–1:10. ACM, New York. doi:10.1145/2463209.2488734
39. Somu Muthukaruppan T, Pathania A, Mitra T (2014) Price theory based power management for heterogeneous multi-cores. SIGARCH Comput Archit News 42(1):161–176. doi:10.1145/2654822.2541974
40. van Stralen P, Pimentel AD (2010) Scenario-based design space exploration of mpsocs. In: Proceedings of IEEE ICCD'10, pp 305–312
41. Vangal S, Howard J, Ruhl G, Dighe S, Wilson H, Tschanz J, Finan D, Singh A, Jacob T, Jain S, Erraguntla V, Roberts C, Hoskote Y, Borkar N, Borkar S (2008) An 80-tile sub-100-w teraflops processor in 65-nm cmos. IEEE J Solid State Circuits 43(1):29–41. doi:10.1109/JSSC.2007.910957
42. Vassiliadis S, Sourdis I (2006) Flux networks: Interconnects on demand. In: International conference on embedded computer systems: architectures, modeling and simulation, 2006. IC-SAMOS 2006, pp 160–167. doi:10.1109/ICSAMOS.2006.300823
43. Vassiliadis S, Wong S, Gaydadjiev G, Bertels K, Kuzmanov G, Panainte EM (2004) The molen polymorphic processor. IEEE Trans Comput 53(11):1363–1375. doi:10.1109/TC.2004.104
44. Wentzlaff D, Griffin P, Hoffmann H, Bao L, Edwards B, Ramey C, Mattina M, Miao CC, Brown JF III, Agarwal A (2007) On-chip interconnection architecture of the tile processor. IEEE Micro 27(5):15–31. doi:10.1109/MM.2007.89
45. Ykman-Couvreur C, Avasare P, Mariani G, Palermo G, Silvano C, Zaccaria V (2011) Linking run-time resource management of embedded multi-core platforms with automated design-time exploration. Comput Digit Tech IET 5(2):123–135. doi:10.1049/iet-cdt.2010.0030